

CMPT 215

Introduction to Computer Organization and Architecture

Assignment Two

Due Date: Friday February 10th, 6:00 pm – late submissions will not be accepted
all submissions MUST use the E-Handin system

Total Marks: 104

1. (12 marks) Perform the following conversions.

(a) $110111_2 = ?_{10}$	(c) $AF2E_{16} = ?_2$	(e) $591_{10} = ?_{16}$
(b) $1111110001101_2 = ?_8$	(d) $4B_{16} = ?_{10}$	(f) $65_7 = ?_{11}$
2. (6 marks) Represent the following signed numbers in 8 bit 2's complement.

(a) -31_{10}	(b) -2_{10}	(c) 123_{10}
----------------	---------------	----------------
3. (6 marks) Evaluate the following expressions, in which each operand is an 8 bit 2's complement number, using the standard binary addition algorithm (evaluate the subtraction in part (a) by negating then adding). As well as showing each result in 8 bit 2's complement, also convert and express each as a signed decimal number. Note that whenever there is signed overflow your answer will not be correct. For which expression(s) does this occur?

(a) $11010101 - 01101100$	(b) $11100001 + 00011111$	(c) $10011001 + 10101010$
---------------------------	---------------------------	---------------------------
4. In this question you are to write some procedures in MIPS assembly language for building and using an undirected graph data structure. Each node of the graph is constrained to have at most three incident edges, and is represented using a data structure consisting of four consecutive words of memory, as shown below:

flags	node ID	node pointer (edge)	node pointer (edge)	node pointer (edge)
address x	address $x+4$	address $x+8$	address $x+12$	

The two highest-order bits in the first word are used for two flags. The “used” flag bit (the highest order bit) will be 1 if the other fields of the node data structure have been filled in with valid values, and 0 otherwise. The next bit, the “visited” flag bit, will be explained below. The low-order three bytes of the first word contain an integer node ID. Following this is one word for each of the three possible incident edges, each storing a node pointer. Each pointer gives the memory address of the *first word* of the data structure for the node at the other end of the edge. If a node has fewer than three incident edges, each unused node pointer is set to zero.

Use “.space 800” to allocate 800 bytes (200 words) of memory. You will use this memory for an array of 50 of the 4-word node data structures described above. In your main program, initialize all 200 words to zero.

For **each** of parts (a)-(d), you will need to hand in the source code of your (appropriately commented) procedure(s) and of a main program that you used for testing. Your documentation for each part should include a description of **what** your code does, **how** it works, what **limitations** (if any) it has, and a precise description of the **testing** you have done. **All of your procedures must use the “standard” conventions discussed in class for passing parameters and returning results.**

- (a) (20 marks) Write a procedure *addnode* that takes as its parameters the starting address of an array of node data structures, the size of the array, and the node ID for a new node. Your procedure should then traverse the array from its beginning, looking at the “used” flag bit in the first word in each node data structure, until one is found with value 0. Your procedure should then set the “used” flag bit to 1, fill in the integer node ID, and store zero into each of the three node pointer fields. Your procedure should then return the value 0 (success). If your procedure traverses the entire array of node data structures without finding an unused one, your procedure should return -1 (failure).
- (b) (20 marks) Write a procedure *deletenode* that takes as its parameters the starting address of an array of node data structures, the size of the array, and a node ID. Your procedure should return 0 if a node was found with that integer ID, and -1 otherwise. In the former case, it should set the “used” flag bit of the corresponding node data structure to 0, thus deleting the node from the graph.
- (c) (20 marks) Write a procedure *addedge* that takes as its parameters the starting address of an array of node data structures, the size of the array, and two node IDs. Your procedure should find the node data structures for these nodes (by traversing the array) and add an edge between them. Your procedure should return 0 if successful (i.e., the two node IDs were found in “used” node data structures, and each had at most two existing incident edges), and otherwise return -1.
- (d) (20 marks) Write a procedure *connected* that determines whether two nodes in the graph are (directly or indirectly) connected. Your procedure should take as its parameters the starting address of an array of node data structures, the size of the array, and two node IDs. Your procedure should first find the node structure for one of these nodes by traversing the array (returning -1 if not found). Then, it should call a **recursive** procedure *search* that takes as its parameters the memory address of the first word of a node data structure (the “starting point” of the search), and a node ID (the node to be found by following edges from the starting point). Your *search* procedure should return 0 if the node ID given as a parameter, matches the node ID in the node data structure whose address is given as a parameter; if not, and if the “visited” flag bit is 1, it should return -1. If neither of these cases hold, *search* should set “visited” to 1, and then should make a recursive call for each node sharing an edge with the starting point node, until one of these calls returns 0, or all edges have been tried. In the former case, *search* should return 0, in the latter case, -1. Your *connected* procedure should return 0 if its call to *search* returns 0, and -1 otherwise. In either case, before returning your *connected* procedure must traverse the array of node data structures, setting each “visited” flag bit back to 0.